# ”VOICE AND GESTURE DEVELOPMENT ENVIRONMENT: KEYBOARD FREE PROGRAMMING”

A Thesis

by

LEANA MORGAN BOUSE

BS, Texas A&M University - Corpus Christi, 2014

Submitted in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Texas A&M University-Corpus Christi
Corpus Christi, Texas

August 2017

”VOICE AND GESTURE DEVELOPMENT ENVIRONMENT:

KEYBOARD FREE PROGRAMMING”

A Thesis

by

LEANA MORGAN BOUSE

This thesis meets the standards for scope and quality of
Texas A&M University-Corpus Christi and is hereby approved.

August 2017

## ABSTRACT

Computer interaction is essential for computer science professionals. Traditional input devices, such as a keyboard and mouse, can be difficult obstacles for professionals with pre-existing manual impairments or may develop future manual impairments as a result of extensive computer use. To address the needs of these professionals, we developed the Voice and Gesture Development Environment (VGDE), allowing users to create Java programs through the use of voice and gesture, by using Microsoft Speech for speech recognition engine and the Leap Motion device and Application Program Interface (API) for gesture recognition. This software allows users to dictate Java code naturally through voice commands, as well as edit and navigate the user's code with voice and gestures. The purpose of the VGDE is to reduce the amount of keyboard and mouse usage while programming and interacting with a development environment.

TABLE OF CONTENTS

CONTENTS PAGE

ABSTRACT . . . v

TABLE OF CONTENTS . . . vi

LIST OF FIGURES . . . vii

LIST OF TABLES . . . viii

CHAPTER I: INTRODUCTION . . . 1

1.1 Prior Works . . . 2

1.2 Problem Description . . . 3

1.3 Contributions . . . 4

1.4 Thesis Overview . . . 4

CHAPTER II: SYSTEM DESIGN . . . 6

2.1 Command System . . . 6

2.2 Software Structure . . . 17

CHAPTER III: EVALUATION AND RESULTS . . . 28

3.1 Evaluation of the Accuracy of Voice Commands . . . 28

3.2 Evaluation of the Accuracy of Gesture Commands . . . 30

3.3 Results of the Evaluations . . . 31

CHAPTER IV: FUTURE WORK . . . 33

CHAPTER V: CONCLUSION . . . 35

REFERENCES . . . 37

APPENDIX A: SEQUENCE DIAGRAMS . . . 39

LIST OF FIGURES

FIGURES PAGE

2.1 Sequence Diagram of Voice Programming Command “ONE” . . . . . . . . . . . . . . 8

2.2 Sequence Diagram of Voice Environment Command “Open File” . . . . . . . . . . . . 13

2.3 Environment Voice Command: Pause Microphone . . . . . . . . . . . . . . . . . . . . . . 14

2.4 Sequence Diagram of Voice Environment Command “Pause Microphone” . . . . . . . 14

2.5 Sequence Diagram of Gesture Navigation Command “Move Cursor Down” . . . . . . 16

2.6 Component Interaction Overview Diagram . . . . . . . . . . . . . . . . . . . . . . . . . 17

2.7 Voice Component Interaction Overview Diagram . . . . . . . . . . . . . . . . . . . . . 19

2.8 Gesture Component Interaction Overview Diagram . . . . . . . . . . . . . . . . . . . . 23

2.9 Parser Component Interaction Overview Diagram . . . . . . . . . . . . . . . . . . . . . 25

2.10 Image of VGDE user interface, with components labeled 1-8. . . . . . . . . . . . . . . 27

3.11 Voice Command Test 1: Convert Integer to Binary . . . . . . . . . . . . . . . . . . . . 29

3.12 Full Test Code - Including headers and template . . . . . . . . . . . . . . . . . . . . . 30

6.13 Sequence Diagram of Voice Programming Command “ONE” . . . . . . . . . . . . . . 39

6.14 Sequence Diagram of Voice Environment Command “Open File” . . . . . . . . . . . . 40

6.15 Sequence Diagram of Voice Environment Command “Pause Microphone” . . . . . . . 41

6.16 Sequence Diagram of Gesture Navigation Command “Move Cursor Down” . . . . . . 42

LIST OF TABLES

TABLES PAGE

2.1 General Command Categories and Subcagetories . . . . . . . . . . . . . . . . . . . . . . 6

2.2 A list of Voice Coding Commands. Subcategories: Data type, Operators, Assignment, and Comparison. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.3 A list of Voice Coding Commands. Subcategories: Selection and Iteration. . . . . . . . . 9

2.4 A list of Voice Coding Commands. Subcategories: New Line and InputOutput. . . . . 9

2.5 A list of Voice Coding Commands. Subcategories: Digits. . . . . . . . . . . . . . . . . . 10

2.6 A list of Voice Editing Commands. Subcategories: Select and Undo. . . . . . . . . . . . 10

2.7 A list of Voice Editing Commands. Subcategories: Delete and Clipboard. . . . . . . . 11

2.8 A list of Voice Navigation Commands. Subcategories: Move cursor. . . . . . . . . . . 12

2.9 A list of Voice Environment Commands. Subcategories: Software and File. . . . . . . 14

2.10 A list of Voice Environment Commands. Subcategories: Microphone and Gesture. . . . 15

2.11 A list of Gesture Editing Commands. Subcategories: Select and Delete . . . . . . . . . 16

2.12 A list of Gesture Navigation Commands. Subcategories: Move cursor. . . . . . . . . . 16

2.13 UML Class Diagram of the Keyword Class . . . . . . . . . . . . . . . . . . . . . . . 20

2.14 UML Class Diagram of the Grammar Class . . . . . . . . . . . . . . . . . . . . . . . 20

3.15 A list of voice commands spoken for evaluation test. . . . . . . . . . . . . . . . . . . . 29

3.16 A list of gesture commands executed for evaluation test. . . . . . . . . . . . . . . . . . 31

3.17 Results of the Voice Command evaluation on microphone and timing accuracy. . . . . 32

3.18 The results of the Gesture Command evaluation on accuracy. . . . . . . . . . . . . . . 32

# CHAPTER I: INTRODUCTION

The professional career of a programmer is reliant on his or her ability to manage physical and mental tasks. Programming, in and of itself, is often a sedentary job; the act of using traditional input devices, such as a keyboard and mouse, to create programs can cause pain or discomfort in manually impaired programmers. Manually impaired programmers are individuals with medical conditions affecting their ability to efficiently or extensively use traditional input devices. Whether a manually impaired programmer has a pre-existing condition, or has developed their condition as a result of their profession, the repetitive task of typing can limit career access for these individuals. While there are commercially available software packages to help these individuals use a computer through speech recognition or simple gestures, there are still severe limitations for programmers falling within this category. In order to provide an aid to programmers with manual impairments, we have developed the *Voice and Gesture Development Environment* (VGDE), a code editor that eliminates or minimizes the use of the keyboard and mouse.

The *VGDE* reduces or eliminates repetitive physical tasks associated with typing and mouse usage by instead allowing the user to create Java programs by voice and gesture recognition. This software allows users to speak in a natural manner to create their program, rather than dictating exact source code. The primary use of the gesture recognition component is to navigate and edit the user's source code, as well as control the softwares interface. Gestures are an optional feature, to ensure they do not limit the usability of the system. For each gesture command, there is a corresponding voice command, so users are able to use the command method aligning with their needs. The primary motivation for developing this code editor is to provide accessibility to manually impaired programmers and to reduce strain or injury. However, the goal is for this solution to appeal to all programmers, not only those who are manually impaired. Providing options and customization allows programmers to find a balance between voice and gesture commands that are best suited for their work patterns.

## 1.1 Prior Works

There have been various approaches to developing alternative means for programming. A number of these solutions include using voice recognition to create programs. While the approaches these systems take may vary, there is always one agreeing factor: expanding the accessibility of programming.

Wichita State University worked on integrating Natural User Interfaces (NUIs) with Integrated Development Environments (IDEs) [5]. Their work uses gesture and voice commands for interacting with the interface for Microsoft Visual Studio, an IDE. The authors' program operates as an independent program from Visual Studio that is executed and run at the same time as working with the IDE. The Microsoft Kinect is used for speech and gesture recognition. The primary goal for the NUI software was not to code a program, but to interact with the IDE interface. While this lessened the use of keyboard and mouse in the programming process, the project faced issues with correctly recognizing speech and gestures. It took an average of two voice command attempts for the system to recognize the correct voice command, and four to five gesture command attempts.

Arnold, Mark, and Goldthwaite designed a system for programming by voice by which a commercially available speech-to-text software, Dragon Naturally Speaking, used in combination with syntax-directed editor generators, to create programs by voice [2]. The main obstacle this system faced was determining what verbal syntax for programming should be used. The designs for this project focused on the combination of the speech recognition software and the syntax-directed editor generators. While there are many potential benefits to this system, it was not ultimately implemented.

Another type of voice programming system was researched by Kumar, Agarwal, and Manwani of IBM's research and software groups [7]. The system allows for creating programs verbally through the use of a telephone, which connects to a program which executes when the call is connected and then terminated when the call is ended. Rather than dictating code, the user is offered various options of components to add to the program they are building. While this may not allow the freedom that traditionally constructed programming offers, it is a good tool for novice programmers.

The Voice-Activated Syntax-Directed Editor (VASDE), developed by Langan, Hain, Hubbell, and Frseth, provides another voice-based program editing system [6]. The framework for VASDE is based on the Eclipse Integrated Development Environment (IDE). This system relies heavily on a graphic user interface for helping the user create a program. The user verbally or manually navigates through menus in the user interface to construct their code. Features of a user's source code, such as identifiers, methods, and statements, are displayed in a list format, rather than the blocks of code seen in traditional source code. Navigation through the source code is accomplished through voice commands, or may be done manually.

Ordonez-Franco, et al., proposed vocabulary and grammar which was outlined for a possible open-source speech recognition programming platform [10]. This research was focused entirely on how the verbal commands should be structured, but suggested future work for a full system for recognizing and parsing commands with this basic vocabulary.

## 1.2 Problem Description

Although the related work has addressed some problems faced by manually impaired programmers, there is still room for improvement. Computer science is a demanding field, not only mentally, but physically. It requires many hours spent typing on a keyboard and navigating code with a mouse. This provides a barrier to people that may be interested in computer science, but are unable to pursue it due to these manual impairments. Long hours of constant use can also cause conditions to develop in those already in the computer science field. A prevailing issue among programmers is carpal tunnel syndrome, especially for those who have used computers for work for over eight years [1].

Navigating and editing code, especially of very large source code files, can cause pain or discomfort to manually impaired programmers. In order to eliminate or greatly reduce the usage of a keyboard and mouse when programming, the gesture recognition portion of the proposed editor is used primarily for navigation and editing purposes. However, as previously mentioned, the editor is designed so that it appeals to all programmers, regardless of whether they are impaired or not. It is not only a tool for accessibility, but for efficiency.

## 1.3 Contributions

We have developed the *Voice and Gesture Development Environment* (*VGDE*). This software has been developed as a Java code editor, which expands on the capabilities of prior works aimed at assisting manually impaired programmers. This editor combines speech and gesture recognition to allow the user to program and edit their source code. The voice commands for generating source code should feel natural, but also should make efficient use of the number of spoken words for each line of source code.

The primary contribution of the *Voice and Gesture Development Environment* is combining voice and gesture to limit the use of keyboard and mouse while programming. The user can program through natural voice commands, rather than strictly dictating the code. The use of filler words while giving voice commands does not interfere with the correct parsing of these commands, provided filler words do not conflict with any existing keywords. Each voice command also has variations of the command, so the user has more flexibility in speaking naturally. Code generated by voice commands can be navigated and edited in the code editor. Navigating and editing can be accomplished by voice commands, gesture commands, or manually with a keyboard and mouse. The user may use any combination of these methods to suit their physical programming needs.

Each gesture command also has a voice command alternative to allow the user to use whichever method of navigating and editing code that suits them best. Voice and gesture commands may also be used to interact with the code editor software itself.

## 1.4 Thesis Overview

This thesis will cover four main topics regarding to the development of the *Voice and Gesture Development Environment*. Chapter two will discuss in depth the system design of the software.

This chapter is broken into two main sections: the Command System and the Software Structure. The Command System section outlines each of the voice and gesture commands that are available in this software. In the second section of chapter two, the Software Structure, it is separated into five subsections: an overview, and each component of the software. The first component covered is the *Voice Component*, which processes words spoken by the user, and then determines which commands the user is issuing. The next component covered is the *Gesture Component*. This component is similar to the first, in that it processes gestures made by the user, and then determines which commands the user is issuing. The third component is the *Parser Component*, which takes the commands interpreted from the previous two components and determines how to execute them. Most commands are executed in the last component, the *Development Environment Component*. This component consists of the user interface, where the results of most commands can be viewed.

The third chapter of the thesis will cover the evaluation and result of the *Voice and Gesture Development Environment*. This chapter is separated into two main sections: the Evaluation, and the Results. The first section discusses the methods that were used in the evaluation of this software. There are two subsections: the Accuracy of Voice Commands and the Accuracy of Gesture Commands. The Results section of this chapter review the aforementioned evaluations and discuss the results.

The fourth chapter discusses the opportunity for future work involving this thesis, and the fifth chapter discusses what conclusions have been made regarding this work.

# CHAPTER II: SYSTEM DESIGN

The *Voice and Gesture Development Environment* (VGDE) is written in the C# programming language with the Microsoft Speech Platform [9] and Leap Motion v. 2.3 [8]. The VGDE system contains three main components. The first component is *Voice*, which manages voice input from a microphone and determines what actions to take based on the user's commands. The *Voice* component allows the user to speak in a manner which is relatively natural while dictating code, controlling the software, or navigating through code.

The second component is *Gesture*, which manages the gesture recognition portion of the software. This component uses gesture recognition to reduce keyboard and mouse use when editing and navigating through code. Each gesture command has a voice counterpart, so the user can freely use whichever is most comfortable or convenient.

The last component is the *Development Environment*, which provides the groundwork supporting the other two components. The Development Environment is the user interface for the software. It contains the code editor, where spoken coding commands are printed to the screen as code, and the basic controls for microphone, Leap Motion device, and other editing features.

## 2.1 Command System

The *Command System* allows users to execute actions in the *VGDE* software by voice and gesture, without having to manually click with a mouse or type on a keyboard. Table 2.1 displays a list of the categories of commands available that the user can execute. Whether a command is voice or gesture, it will fall under one of these categories and subcategories.

Table 2.1: General Command Categories and Subcagetories

| Category | Subcategories |
|---|---|
| Code | Data Type, Operators, Assignment, Comparison, Selection, Iteration |
| Edit | Delete, Clipboard |
| Navigation | Move cursor |
| Environment | Program, File, Microphone, Leap Motion |

## Voice Commands

The user can use voice commands to control many aspects of the system. Four categories of voice commands are available to users: code commands, editing commands, navigation commands, and environment commands. Many of the commands have alternative wording to allow the user a more natural manner of speech. Superfluous words are filtered out, so the user is able to include filler words while speaking to the *VGDE* software without effecting the command recognition, provided that the filler words are not part of any other existing command. For example, the words “um”, “uh”, and “the” would not affect command parsing, however “then”, “one”, and “or” might.

### Code Commands

The main feature of the *Voice and Gesture Development Environment* is the ability to create source code through an advanced speech to text system. The user can dictate code which will appear in the source code file. The code is generated in a textbox in the user interface, where the user can manually edit or add to the source code as needed. Although minimizing keyboard and mouse usage as much as possible is ideal, it is essential to allow the user to have complete control over in which they are working with. This ensures the *VGDE* software is versatile. Figure 2.1 illustrates how a code voice command is processed by the software.

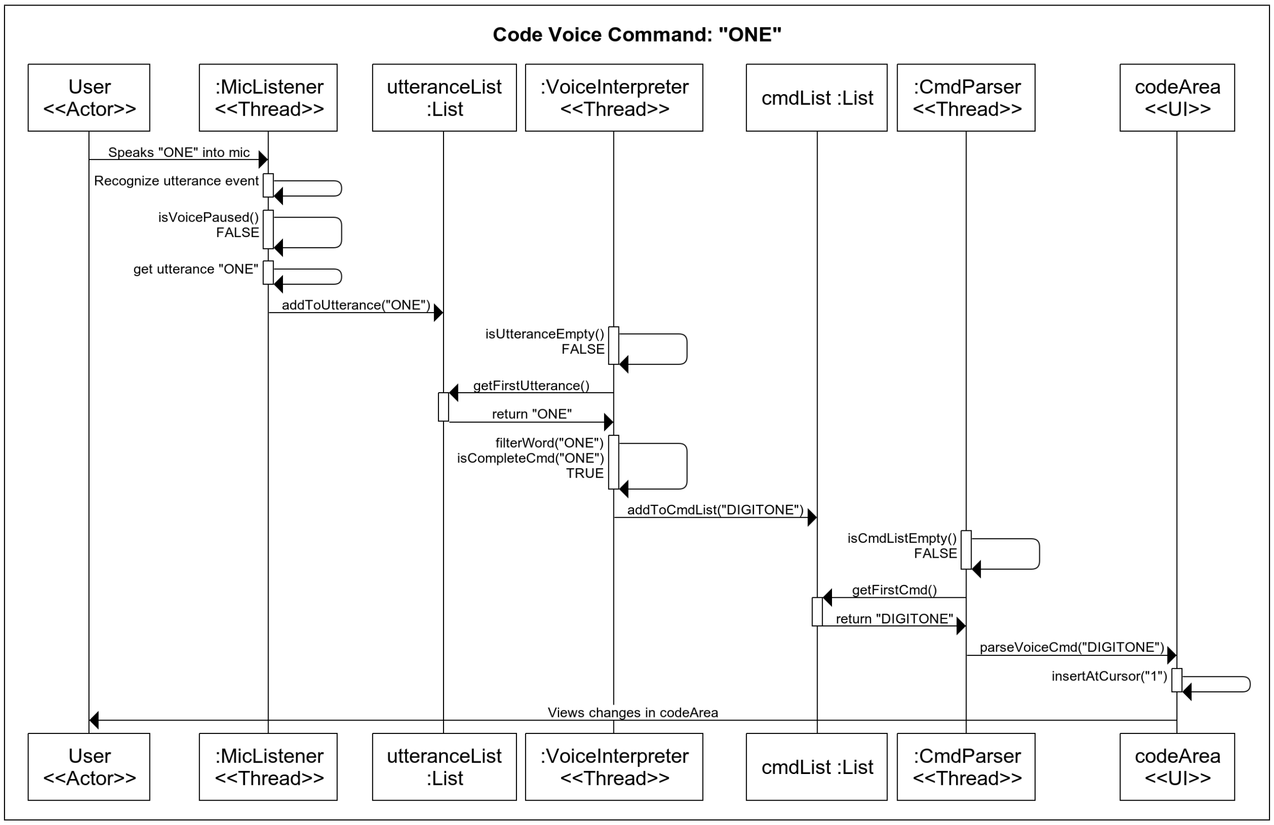


Figure 2.1: Sequence Diagram of Voice Programming Command “ONE”

Tables 2.2, 2.3, 2.4, and 2.5 illustrate the coding commands available to the user, the voice command variations, and the code printed by the command. Some of the code structures, such as iterative and selection code, have particular ways in which they are parsed from command-to-code.

Table 2.2: A list of Voice Coding Commands. Subcategories: Data type, Operators, Assignment, and Comparison.

| Category | Keyword | Utterance | Output |
|---|---|---|---|
| Data Type | Integer | Integer, int, I N T | `int` |
| | Character | Character, char, C H A R | `char` |
| | Float | Float, F L | `float` |
| Operators | Plus | Plus, addition, added | `+` |
| | Minus | Minus, subtracted | `−` |
| | Multiply | Multiply, multiplied, times | `*` |
| | Divide | Divde, divided by | `/` |
| Assignment | Equals | Equals | `=` |
| Comparison | Equal_to | Is equal to, equal to | `==` |
| | And | And | `&&` |
| | Or | Or | `\|\|` |

Table 2.3: A list of Voice Coding Commands. Subcategories: Selection and Iteration.

| Category | Keyword | Utterance | Output |
|---|---|---|---|
| Selection | If | If, Start if | `if(` |
| | Then | Then | `){\n` |
| | EndIf | End if, close if | `\n}\n` |
| | Else | Else | `else {\n` |
| | ElseIf | Else if, start else if | `else if(` |
| | EndElse | End else, close else | `\n}\n` |
| Iteration | WhileLoop | While, Start while | `while(` |
| | WhileDo | Do | `){\n` |
| | EndWhile | End whlie, close while | `\n}\n` |

Table 2.4: A list of Voice Coding Commands. Subcategories: New Line and InputOutput.

| Category | Keyword | Utterance | Output |
|---|---|---|---|
| New Line | Semicolon | Okay, ok, next, semicolon | `;\n` |
| | NewLn | Return, New line | `\n` |
| InputOutput | GetInt | Get int, get integer | `S.nextInt()` |
| | GetChar | Get char, get character | `S.next().charAt(0)` |
| | PrintLn | Print line | `System.out.println(` |
| | Print | Print line | `System.out.print(` |
| | EndPrint | End print | `);\n` |

Table 2.5: A list of Voice Coding Commands. Subcategories: Digits.

| Category | Keyword | Utterance | Output |
|---|---|---|---|
| Digit | Zero | Zero, oh | 0 |
| | One | One | 1 |
| | Two | Two | 2 |
| | Three | Three | 3 |
| | Four | Four | 4 |
| | Five | Five | 5 |
| | Six | Six | 6 |
| | Seven | Seven | 7 |
| | Eight | Eight | 8 |
| | Nine | Nine | 9 |
| | Ten | Ten | 10 |

### Edit Commands

Although editing commands can be accomplished by three means (voice, gesture, or manually), the primary method is by voice. There are three main editing categories of commands: selecting text, deleting text, and clipboard commands. When selecting text, the user can specify the range of lines to select. This selection of text can then be deleted, unselected, or have a clipboard command used on it. The clipboard commands are cut, copy, and paste. These commands access the computer's clipboard, so that the user may copy text from another program and paste it into the *VGDE* or copy from the *VGDE* and paste into another program. Tables 2.6 and 2.7 show each command available, as well as the varied voice commands for each.

Table 2.6: A list of Voice Editing Commands. Subcategories: Select and Undo.

| Category | Keyword | Utterances | Actions |
|---|---|---|---|
| Select | SelLineNum | Select line, select line number | Select line number __ |
| | SelCurr | Select current line, select current | Select the current line |
| | SelAll | Select all | Select all text |
| | Unselect | Unselect, Deselect | Unselect any selected text |
| Undo | Undo | Undo | Undos the last action |
| | Redo | Redo | Redos the last action |

Table 2.7: A list of Voice Editing Commands. Subcategories: Delete and Clipboard.

| Category | Keyword | Utterances | Actions |
|---|---|---|---|
| Delete | Delete all | Delete all, Delete everything, clear | Delete all text |
| | DelSel | delete selection, delete selected | Delete the selected text |
| | DelLine | Delete line, delete current | Delete the current line |
| Clipboard | Copy | Copy, Copy selection | Copies selected text to clipboard |
| | Cut | Cut, Cut selection | Cuts selected text to clipboard |
| | Paste | Paste, Paste text | Pastes text from clipboard to cursor location |

**Navigation Commands**

The navigation commands are primarily intended to be executed through gesture, however, there are also voice commands available to ensure users are able to use whichever means are most suited to them. There is one main type of navigation commands: moving the cursor in the code pane. This command has several specifications so that the user can state where the cursor should be moved to. Table 2.8 displays the voice commands for moving the cursor.

Table 2.8: A list of Voice Navigation Commands. Subcategories: Move cursor.

| Category | Keyword | Utterances | Actions |
|---|---|---|---|
| Move Cursor | MovCurUpLn | move cursor up, move cursor up _ line, move cursor up _ lines | Move cursor up _ line(s) |
| | MovCurDwnLn | move cursor down, move cursor down _ line, move cursor down _ lines | Move cursor down _ line(s) |
| | MovCurHead | move cursor to front of line, move cursor to front of line move cursor to front, move cursor to head | Move cursor to front of the current line |
| | MovCurEnd | move cursor to end of line | Move cursor to end of the current line |
| | MovCurTop | move cursor to top, move cursor to beginning | Move cursor to the begining of the document |
| | MoveCurBottom | move cursor to bottom | Move cursor to the end of the document |

### Environment Commands

The environment commands for the *VGDE* are for controlling the development environment. The user may open, close, and save the source code file they are currently working on. Tables 2.9 and 2.10 list the environment voice commands available to the user. Only one file may be open at a time in the program, however future versions may support multiple files opened in different tabs. The current version of the program supports Java file types, as well as plain text files. Figure 2.2 illustrates the process of opening a file by voice command.

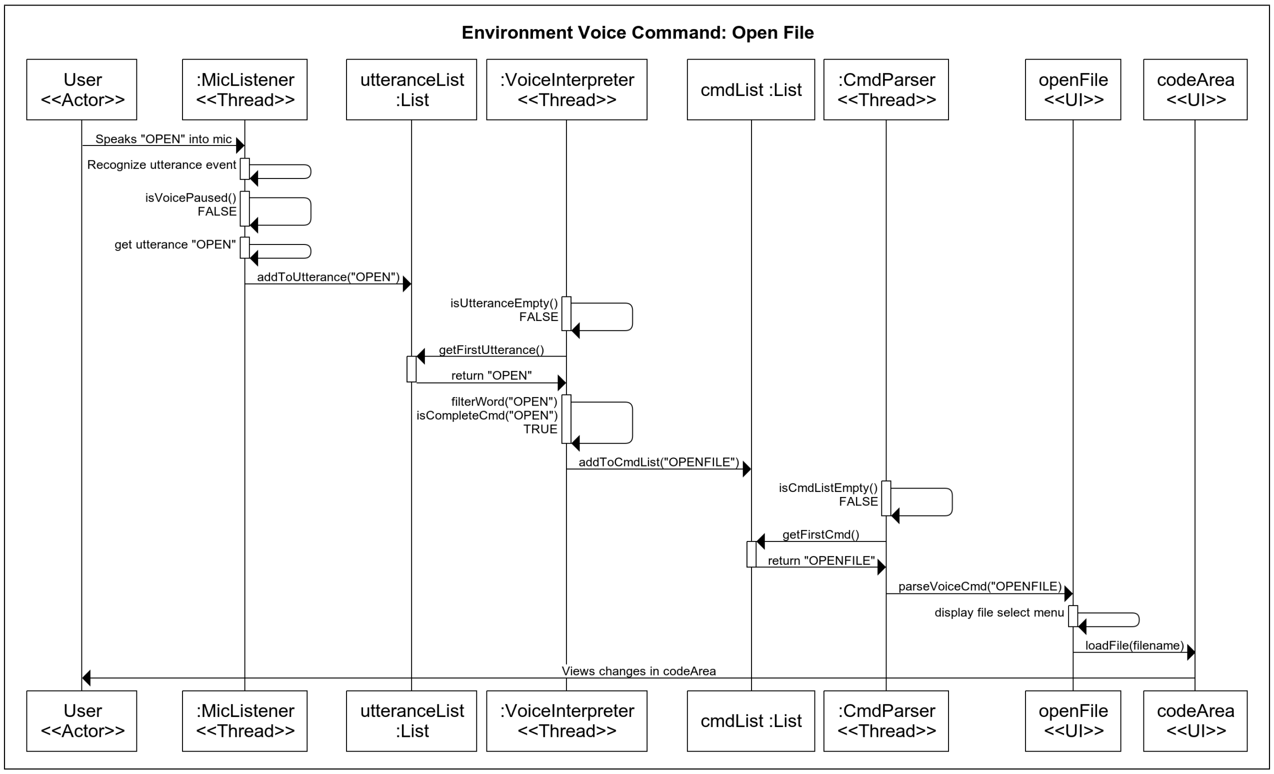


Figure 2.2: Sequence Diagram of Voice Environment Command “Open File”

The user may also use voice commands to control the voice or gesture recognition. The user can choose to pause or mute the microphone for a period of time, which disables any processing of sound from the microphone until it is resumed or unmuted. Likewise, the user can pause gesture recognition through the Leap Motion device. Figure 2.4 demonstrates the process for pausing the microphone by voice command.

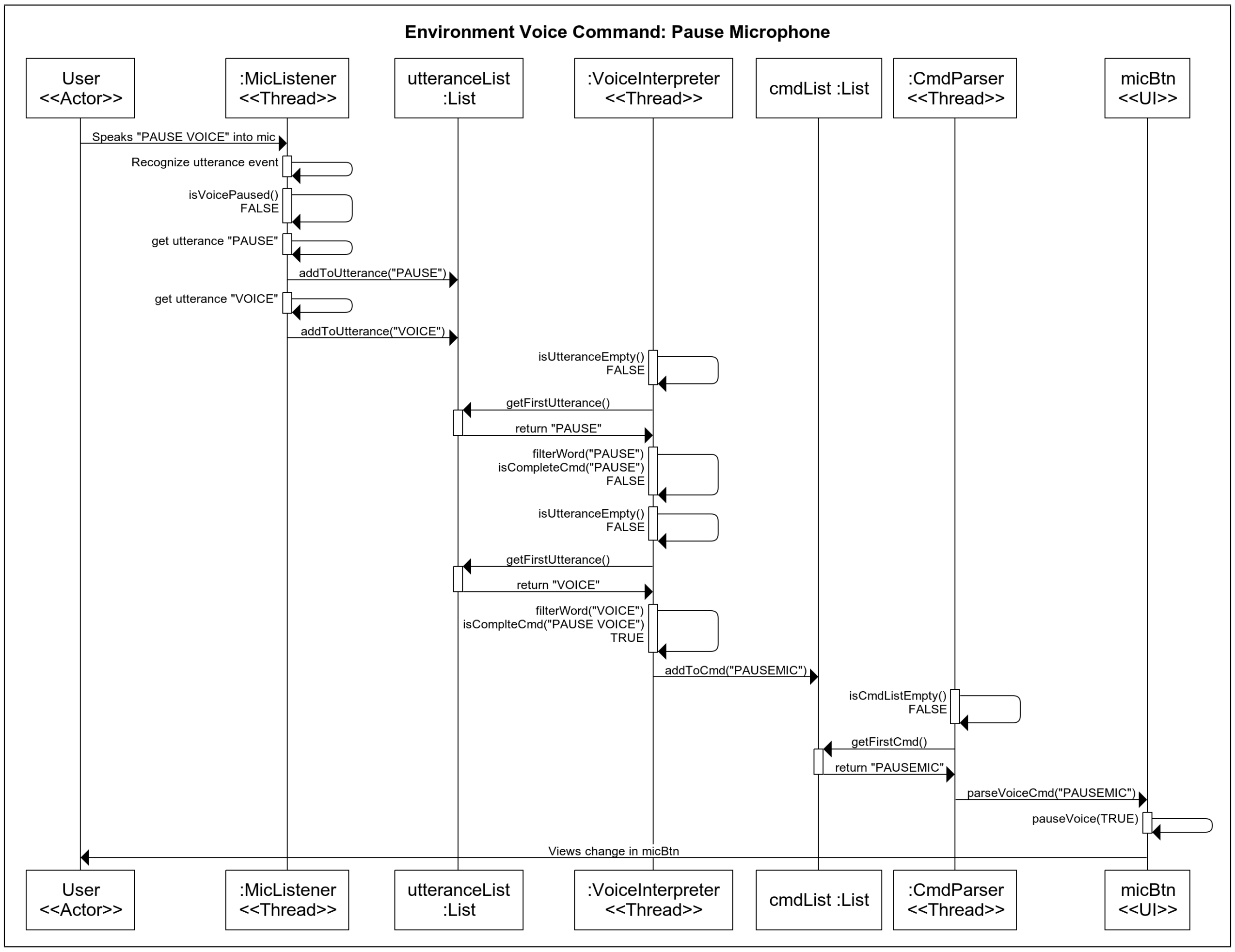


Figure 2.4: Sequence Diagram of Voice Environment Command “Pause Microphone”

Table 2.9: A list of Voice Environment Commands. Subcategories: Software and File.

| Category | Keyword | Utterances | Actions |
|---|---|---|---|
| Software | Exit | exit program, close program | Exits the VGDE Software |
| File | NewFile | new file, create new file, open new file | Creates a new, blank file |
| | OpenFile | open file | Opens the file browser so the user can open a new file |
| | SaveFile | save file | Saves currently opened file |
| | SaveFileAs | save file as, save copy of file | Saves the current document as a new file |
| | CloseFile | close file, close document | Closes current document |

Table 2.10: A list of Voice Environment Commands. Subcategories: Microphone and Gesture.

| Category | Keyword | Utterances | Actions |
|---|---|---|---|
| Microphone | PauseVoice | pause microphone, microphone off, pause mic, pause voice, mute microphone, mute voice | Sets voice to OFF |
| | ResumeVoice | resume microphone, microphone on, resume mic, mic on, resume voice, voice on, unmute micrphone, unmute voice | Sets voice to ON |
| Gesture | PauseGesture | pause gesture, pause leap motion, gesture off | Sets gesture to OFF |
| | ResumeGesture | resume gesture, resume leap motion, gesture on | Sets gesture to ON |

## Gesture Commands

Another important feature of the *VGDE* software is the *Gesture Component*. There are several editing and navigation commands executable through gestures. Although the primary focus of gestures is to reduce mouse usage when navigating through user-generated source code, the user may also edit the code to a limited extent. Tables 2.11 and 2.12 describes several basic edit gesture commands. The specific details about each gesture are the primary means of determining which command is being referenced. For example, the basic loop gesture has multiple variations based on differences in the motion itself, such as the diameter of the loop, direction of the loop, or the loop speed. Figure 2.5 illustrates how gesture commands are processed.

Table 2.11: A list of Gesture Editing Commands. Subcategories: Select and Delete

| Category | Keyword | Gesture | Actions |
|---|---|---|---|
| Select | SelCurr | Swipe Left-Right: 1-Finger | Select the current line |
| | UnSel | Swipe Right-Left: 1-Finger | Unselect text |
| Delete | DelSel | Swipe Down: 1-Finger | Delete selected text |
| | DelCurrLn | Swipe Down: 3-Fingers | Delete the current line |

Table 2.12: A list of Gesture Navigation Commands. Subcategories: Move cursor.

| Category | Keyword | Gesture | Actions |
|---|---|---|---|
| Move Cursor | MovCurUpLn | Counter-Clockwise Circle | Moves cursor up one line |
| | MovCurDwnLn | Clockwise Circle | Moves cursor down one line |
| | MovCurHead | Swipe Right-Left: 2-Fingers | Moves cursor to the front of the current line |
| | MovCurEnd | Swipe Left-Right: 2-Fingers | Moves cursor to end of the current line |

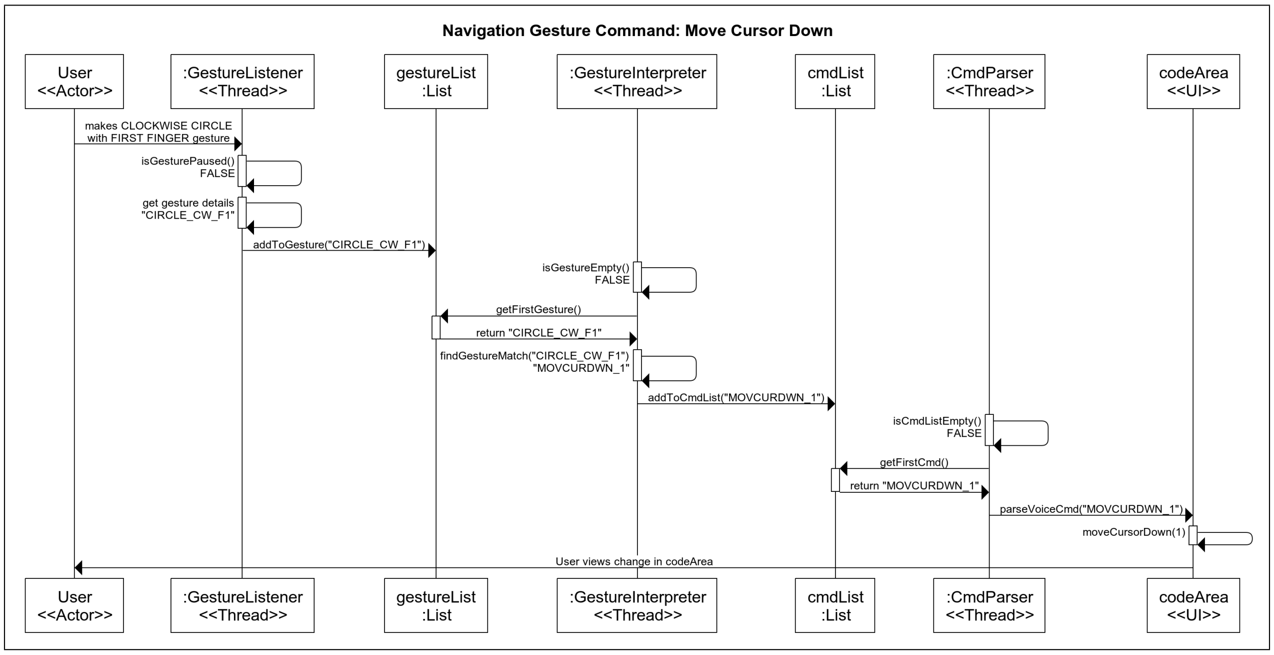

Navigation Gesture Command: Move Cursor Down
User <<Actor>>
:GestureListener <<Thread>>
gestureList :List
:GestureInterpreter <<Thread>>
cmdList :List
:CmdParser <<Thread>>
codeArea <<UI>>
makes CLOCKWISE CIRCLE with FIRST FINGER gesture
isGesturePaused() FALSE
get gesture details "CIRCLE_CW_F1"
addToGesture("CIRCLE_CW_F1")
isGestureEmpty() FALSE
getFirstGesture()
return "CIRCLE_CW_F1"
findGestureMatch("CIRCLE_CW_F1") "MOVCURDWN_1"
addToCmdList("MOVCURDWN_1")
isCmdListEmpty() FALSE
getFirstCmd()
return "MOVCURDWN_1"
parseVoiceCmd("MOVCURDWN_1")
moveCursorDown(1)
User views change in codeArea

Figure 2.5: Sequence Diagram of Gesture Navigation Command “Move Cursor Down”

## 2.2 Software Structure

The software structure of the *VGDE* consists of several threads which operate within four main components: *Voice*, *Gesture*, *Parser*, and *Development Environment* components. Figure 2.6 displays an overview of each of the components and how they interact with one another. Each of these components is spawned from the *Main Thread*. The *Voice* and *Gesture* components are responsible for receiving data from their relative devices and interpreting that data into commands that the user is trying to execute. Once these components have interpreted these commands, the commands are read by the *Parser Component* in chronological order, regardless of whether the command is voice or gesture. The *Parser Component* determines how to execute each command. The majority of the commands executed can be seen in changes in the *Development Environment Component*. The outcome of these components working together is that the user is able to give voice or gesture commands interchangeably, which are interpreted and executed in the *VGDE* software.

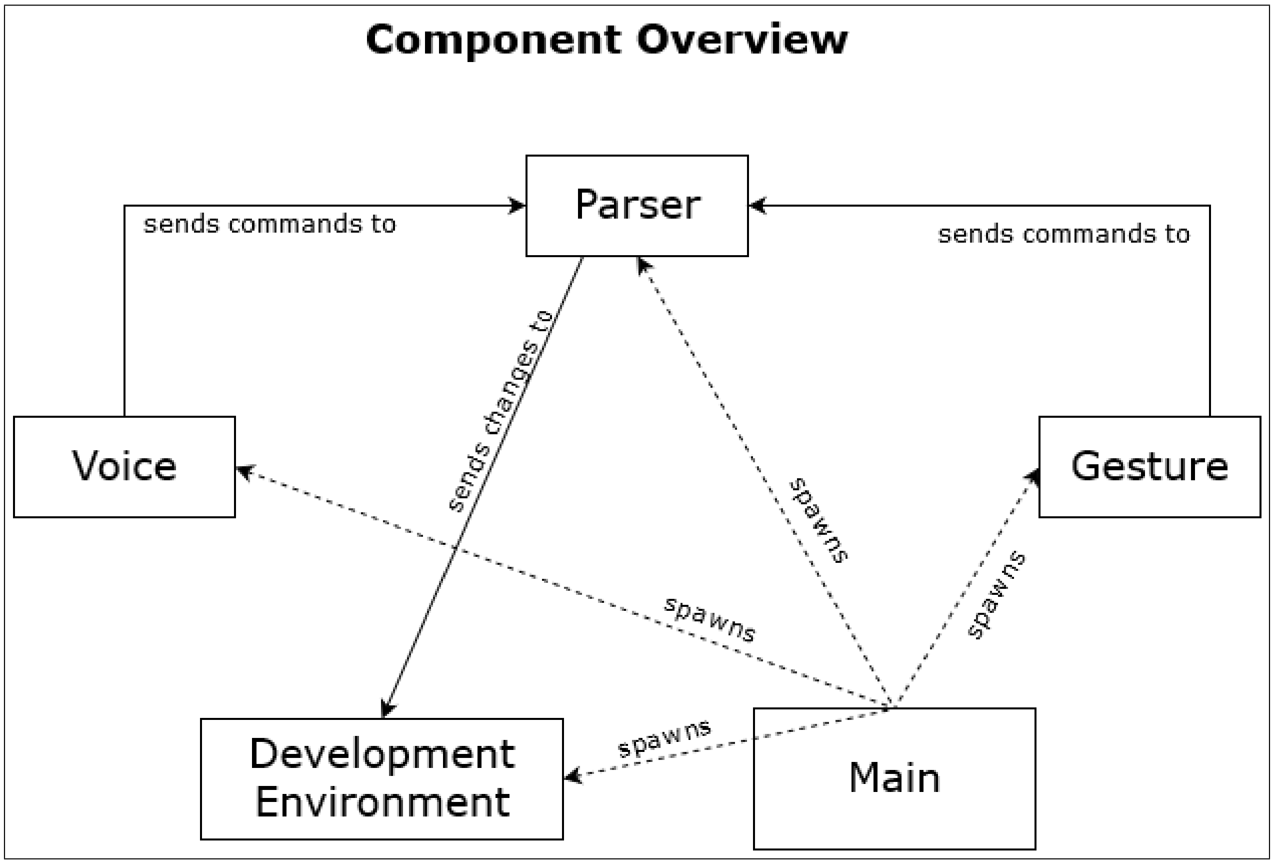


Figure 2.6: Component Interaction Overview Diagram

## Thread Overview

There are six main threads that run asynchronously in the *VGDE* system. These threads interact with three shared Lists. There are two threads for the *Voice Component*, two threads for the *Gesture Component*, one thread for the *Parser Component*, and the *Main Thread*.

The basic process of the *Voice Component* begins with the *Microphone Listener Thread*. This thread takes all spoken words recognized by Microsoft Speech and appends each to the end of a shared *Utterance List*. Then, the *Voice Interpreter Thread* takes the first word of the *Utterance List* and determines if the word is filler, or possibly part of a Keyword. If it is part of a Keyword, it adds the word to another shared list, the *Word List*. If it is a filler, it ignores it. The *Voice Interpreter Thread* then takes each word in the *Word List*, compares it to other words recently spoken and determines what command the user is trying to execute. The interpreted command is then added to a final shared list, the *Command List*. Finally, the *Command Parser Thread* takes each command in the *Command List* and determines how to execute it.

The process of the *Gesture Component* works similar to the *Voice Component*. The *Gesture Listener* waits for a valid gesture to be recognized by the Leap Motion device. The *Gesture Listener* adds the gesture information to the *Gesture Interpreter Thead*. The *Gesture Interpreter* determines what command should be executed. It adds the appropriate command to the *Command List*, which is then executed by the *Command Parser Thread*.

## Voice Component

The *Voice Component* controls many parts of the software. The primary function is to create source code from the user's spoken commands, however the user is also able to navigate and edit their source code, as well as vocally control the software itself. The speech recognition engine used in this project is the Microsoft Speech Platform [9]. Figure 2.7 displays an overview of how the threads interact within this component.

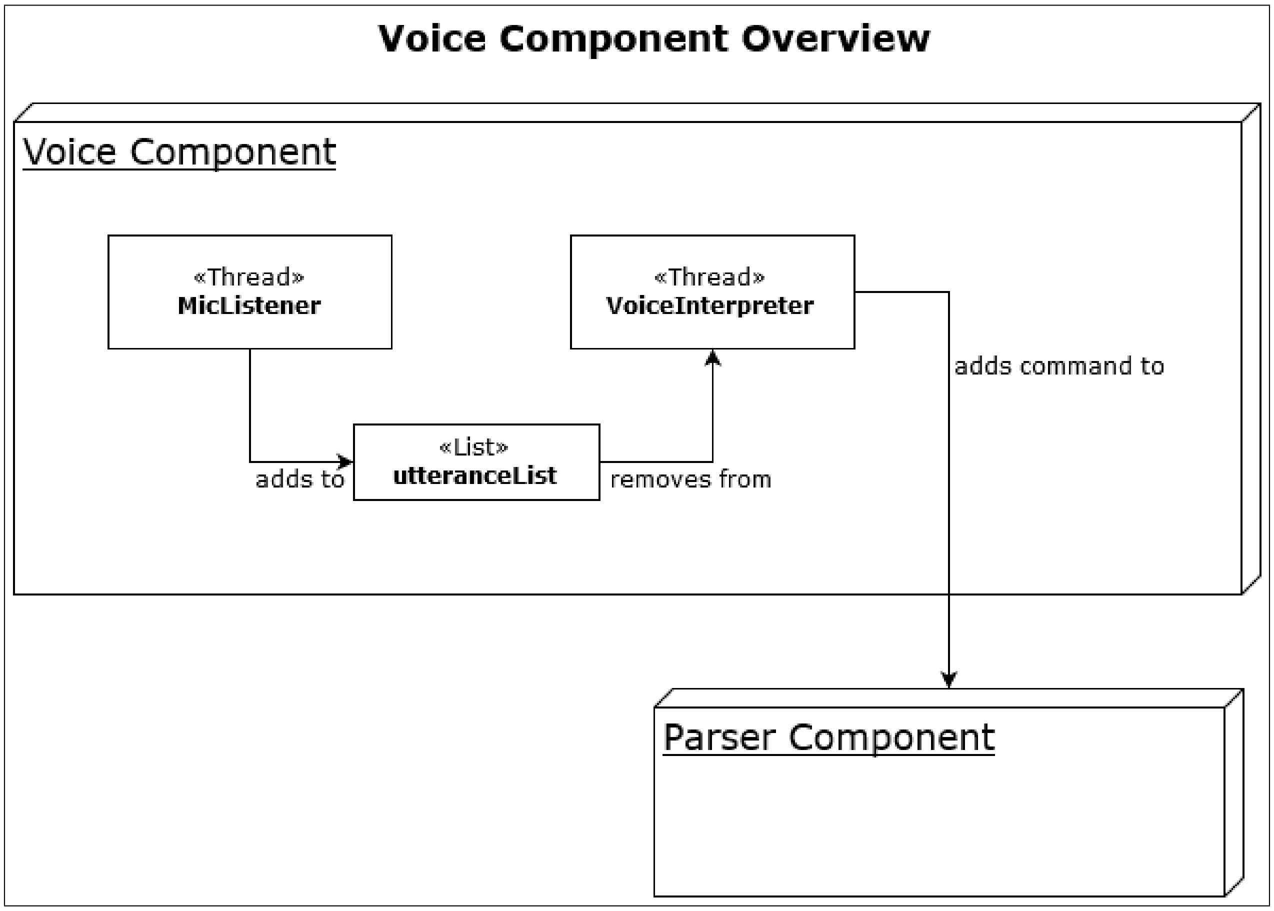


Figure 2.7: Voice Component Interaction Overview Diagram

**Keywords and Grammar Class**

One of the main parts that makes the software work are *Keywords*. *Keyword* is a class that organizes voice and gesture commands. Each step of the *VGDE* utilizes *Keywords*. The *Keyword* class keeps track of the type, category, and whether the keyword is voice or gesture. This helps to organize the processing of *Keywords*. Table 2.13 displays the class diagram for the *Keyword* class. Below this is table 2.14, which displays the class diagram for the *Grammar* class. The *Grammar* class organizes lists of *Keywords* for the *VGDE* to search through when interpreting commands.

Table 2.13: UML Class Diagram of the Keyword Class

| Keyword | |
|---|---|
| − commandType | : String |
| − category | : String |
| − type | : String |
| − keyword | : String |
| − utterances | : String[][] |
| − output | : String |
| − multiWord | : bool |
| + Keyword() | |
| + containsWord() | |

Table 2.14: UML Class Diagram of the Grammar Class

| Grammar | |
|---|---|
| + identifierList | : List<Keyword > |
| + keywordList | : List<Keyword > |
| + Grammar() | |
| + loadIdentifiers() | |
| + loadGestureKeywords() | |
| + loadMuteKeywords() | |
| + loadKeywords() | |
| + findIdentifier() | |
| + getKeyword() | |
| + getKeywordAt() | |
| + findOccurrencesOf() | |
| + getNextMatch() | |

*Keywords* also store the valid *Utterances* for that particular *Keyword*. For example, to output ">=" to the source code file, the user can say it three different ways: "greater than or equal to," "greater than equal to," or "greater than or equal to." In the utterances array, these phrases are stored with each word a separate string.

For *Keywords* that output code to the code area in the User Interface, the text output for that *Keyword* is stored. Additional formatting and processing may be necessary for some *Keywords* when parsing the command, but for easy storage and reference of the code output, this method proved to be the most efficient.

### Microphone Listener Thread

The *Microphone Listener Thread* is the first thread in the voice command pipeline, and its role is relatively simple. It waits until a word is spoken, then it takes the word and adds it to a list. By ensuring each word is spoken is added to a queue as soon as it is spoken and recognized, there is less time wasted in converting spoken words to actions in the program.

This thread is spawned by the *Main Thread*, and remains in an idle state until two conditions are met: a speech recognized event occurs, or the program exits. When the user speaks, Microsoft Speech processes the sound and determines the most likely word that was spoken. The *Microphone Listener Thread* then takes this recognized word and adds it to the *Utterance List*. This is a list which is shared between the *Microphone Listener Thread* and the *Voice Interpreter Thread*. After completing this action, the thread returns to its original, idle state.

### Voice Command Interpreter Thread

The next thread in the voice command pipeline is the *Voice Interpreter Thread*. The purpose of this thread is to filter the spoken words from the *Microphone Listener Thread* and interpret what commands are being issued by the user. This thread utilizes the *Keywords* class previous described, as well as *prevWords*, which is a private list for keeping track of words recently spoken which do not yet make a full command.

The *Voice Interpreter Thread* is spawned by the *Main Thread*, and begins in an idle state. The thread waits until the *Utterance List* is not empty, and then removes the first item from the list and processes it. First, the word must be filtered. Depending on whether the *Voice Paused* flag is true or not, the word may be filtered differently. If the *Voice Paused* flag is true, in other words if voice commands are currently paused or muted, then the word will be compared to the *Voice Paused Commands* list. If the *Voice Paused* flag is false, then the word will be compared to all available voice commands. Only words which are part of at least one of the possible voice commands continue in the interpreting process.

After the recognized speech has been filtered, the *Utterance List* needs to be processed to determine what commands the user is trying to execute. One of the difficulties with determining what command is intended is the occurrence of multi-word commands which contain shorter commands. For instance, the command ”greater than or equal to” for outputting the code “>=” includes three shorter commands: “greater than”, “or”, and “equal to”. Incorrectly interpreted, this could incorrectly output “>||=” to the source code. The *Voice Interpreter Thread* compares previously spoken words to the current word to determine what to output.

To begin the interpretation process, the thread takes the word that has just been filtered and first determines if it is an existing identifier. If it is not, then it determines if the word is a one-word, unique *Keyword*. These are *Keywords* with an utterance variation consisting of only one word, which do not occur in any other utterance of any other *Keyword*. For example, the word “zero” is a one-word, unique *Keyword*. There are no other occurrences of the word “zero” in any other *Keyword*. If this check is true, then the command is pushed onto a stack called the *addStack*. After this, the thread checks the *prevWords* list and determines if the combined words equal to a complete command. If the contents of *prevWords*are a complete command, then the command is pushed the *addStack*. If the contents are not a complete command, then the *prevWords* list is cleared and the words discarded.

If the word is not a one-word, unique *Keyword*, then the thread determines if the current word, when added to the *prevWords* list, occurs in any existing *Keyword*. If the combination of utterances is a complete command which occurs only once in the possible *Keywords*, it is pushed to the *addStack*. If the combination occurs, but is not a complete and unique command, then the this portion of the filtering process is complete. If a combination does not occur, and the *prevWords* list without the current word is not a complete command on its own, then the *prevWords* list is cleared and the current word is added alone. If the combination does not occur, but the *prevWords* list without the current word is a complete command on its own, then that command is pushed onto the *addStack*. The *prevWords* list is then cleared and the current word is added alone.

The last task that the *Voice Interpreter Thread* completes is to pop any entries in the *addStack* (First-In, Last-Out) and to add those commands to the *Command List*. The *cmdList* is a list shared among the *Voice Interpreter Thread*, *Gesture Interpreter Thread*, and the *Command Parser Thread*. It stores both voice and gesture commands that need to be processed, in the order in which they are inputted by the user. After this is complete, the *Voice Interpreter Thread* returns to its idle state.

### Gesture Component

The gesture recognition device that is used in this project is the Leap Motion, using the 2.3 SDK. This device is very useful for small, minute gestures at a close range. The sensor range of the Leap Motion is two feet above and two feet on each side. The device's small size allows it to be placed between the user and the keyboard. It can recognize the fingers of each hand, and has several simple, recognizable hand gestures. The Leap Motion's API can create custom gestures and optimize the recognition of default and custom gestures. Figure 2.8 displays an overview diagram of how the threads within this component interact with one another.

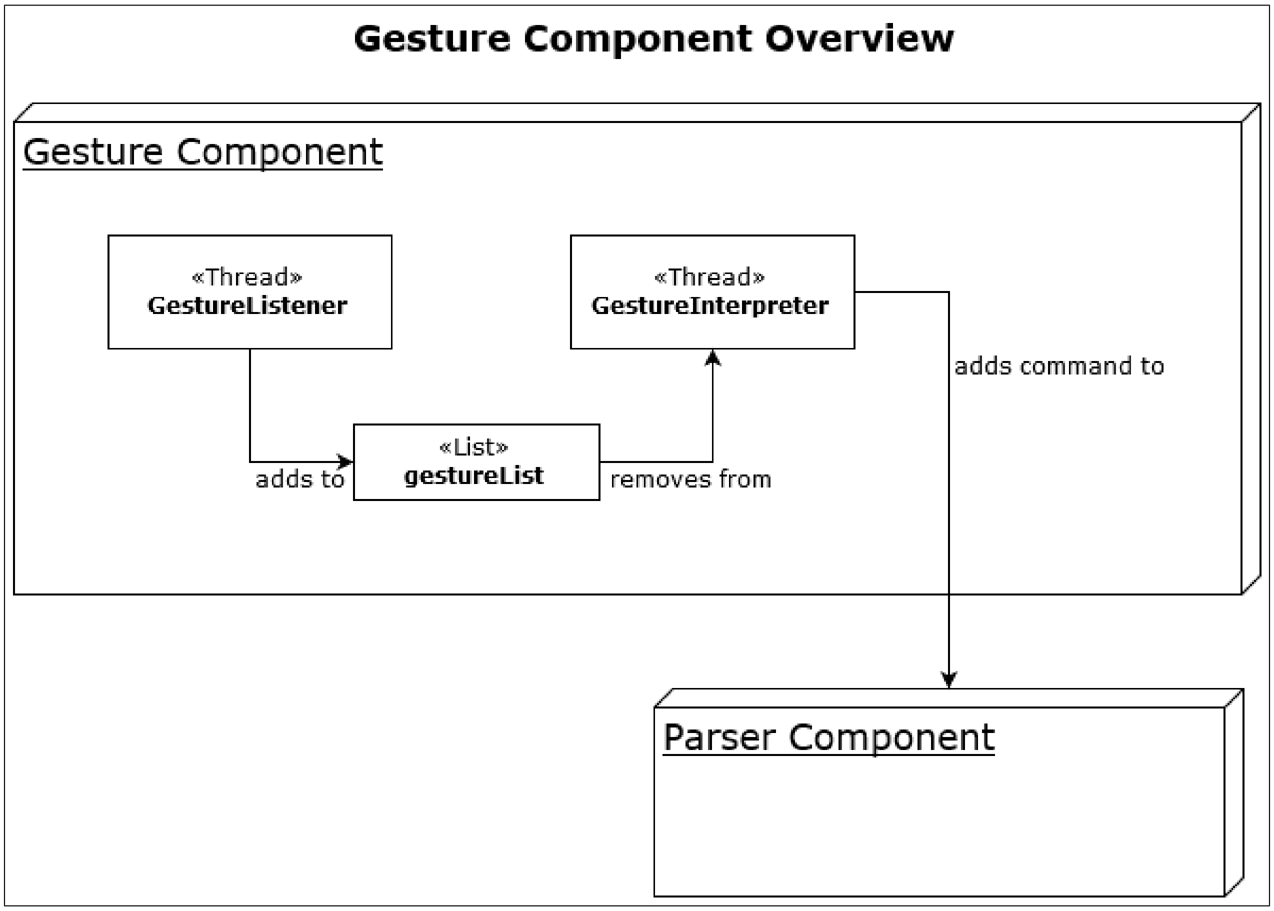


Figure 2.8: Gesture Component Interaction Overview Diagram

### Gesture Listener Thread

The first thread in the gesture command pipeline is the *Gesture Listener Thread*. It is spawned by the *Main Thread*, and begins in an idle state. The thread waits until a new gesture has been recognized by the Leap Motion API. Once a gesture has been detected, the gesture and its specifications are added to the end of the *Gesture List*. There is no need to filter gestures, as the gesture recognition API does not recognize a gesture until it is valid.

### Gesture Interpreter Thread

The next thread is the *Gesture Interpreter Thread*, which is also spawned by the *Main Thread*. This thread begins in an idle state, and remains so until the *Gesture List* is not empty. Once there is at least one item in the *Gesture List*, the *Gesture Interpreter Thread* removes the first item from the list. The thread compares the gesture and gesture specifications with the available gesture commands. The most accurate match for the gesture is determined, and the matching command is then added to the end of the shared *Command List*. The process for interpreting gesture commands is less complex than voice commands, because there are less complex possible commands.

## Parser Component

The third main component of the *VGDE* is the *Parser Component*. This component consists of only one thread and one list, which is the *Command List*. The purpose of this component is to execute the commands which have been passed to it from the *Voice Component* and *Gesture Component*. Most of the commands that are executed can be seen through changes in the *Development Environment*. Figure 2.9 displays an overview diagram of the thread within this component and how it interacts with the rest of the software.

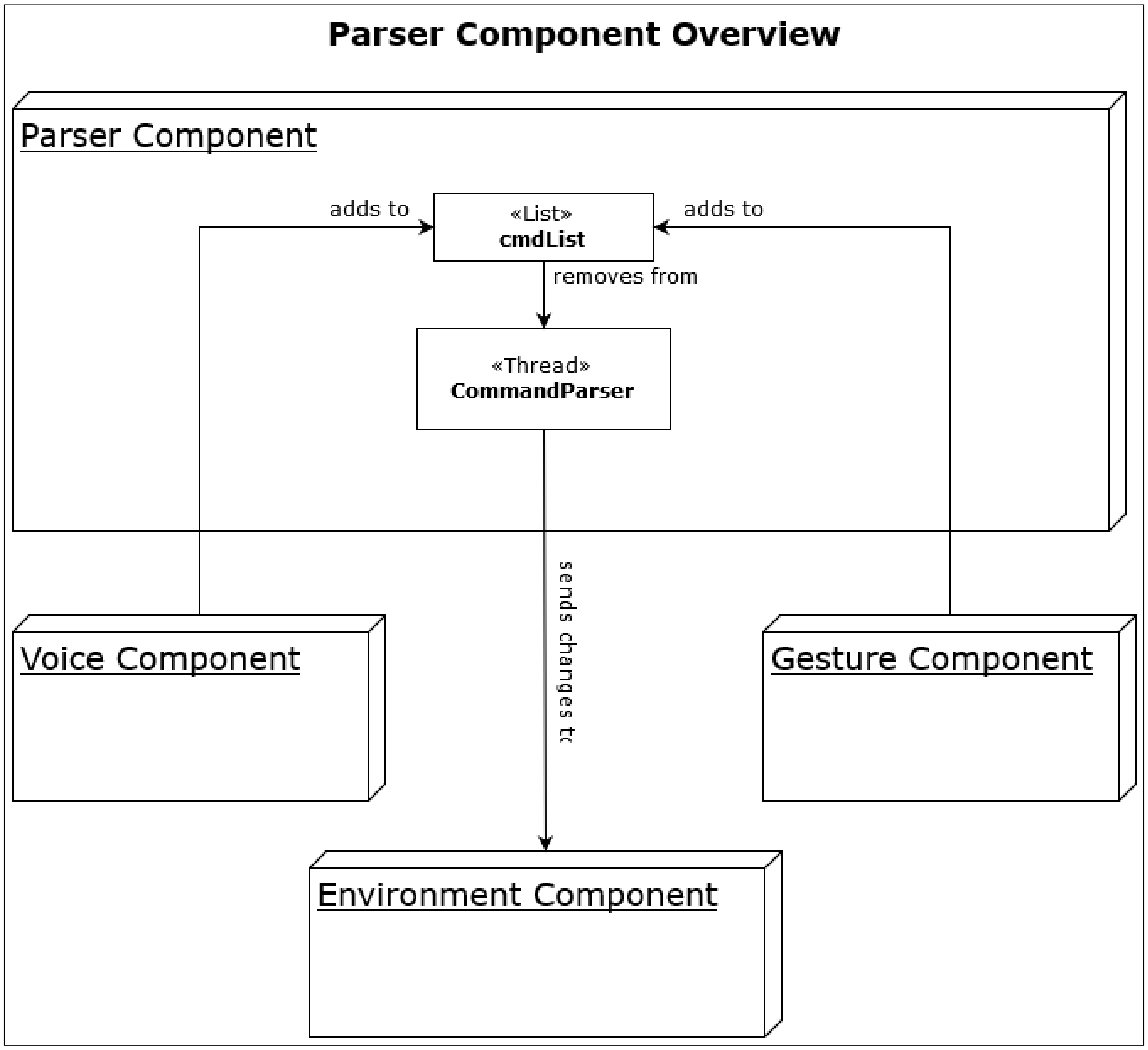


Figure 2.9: Parser Component Interaction Overview Diagram

**Command Parser Thread**

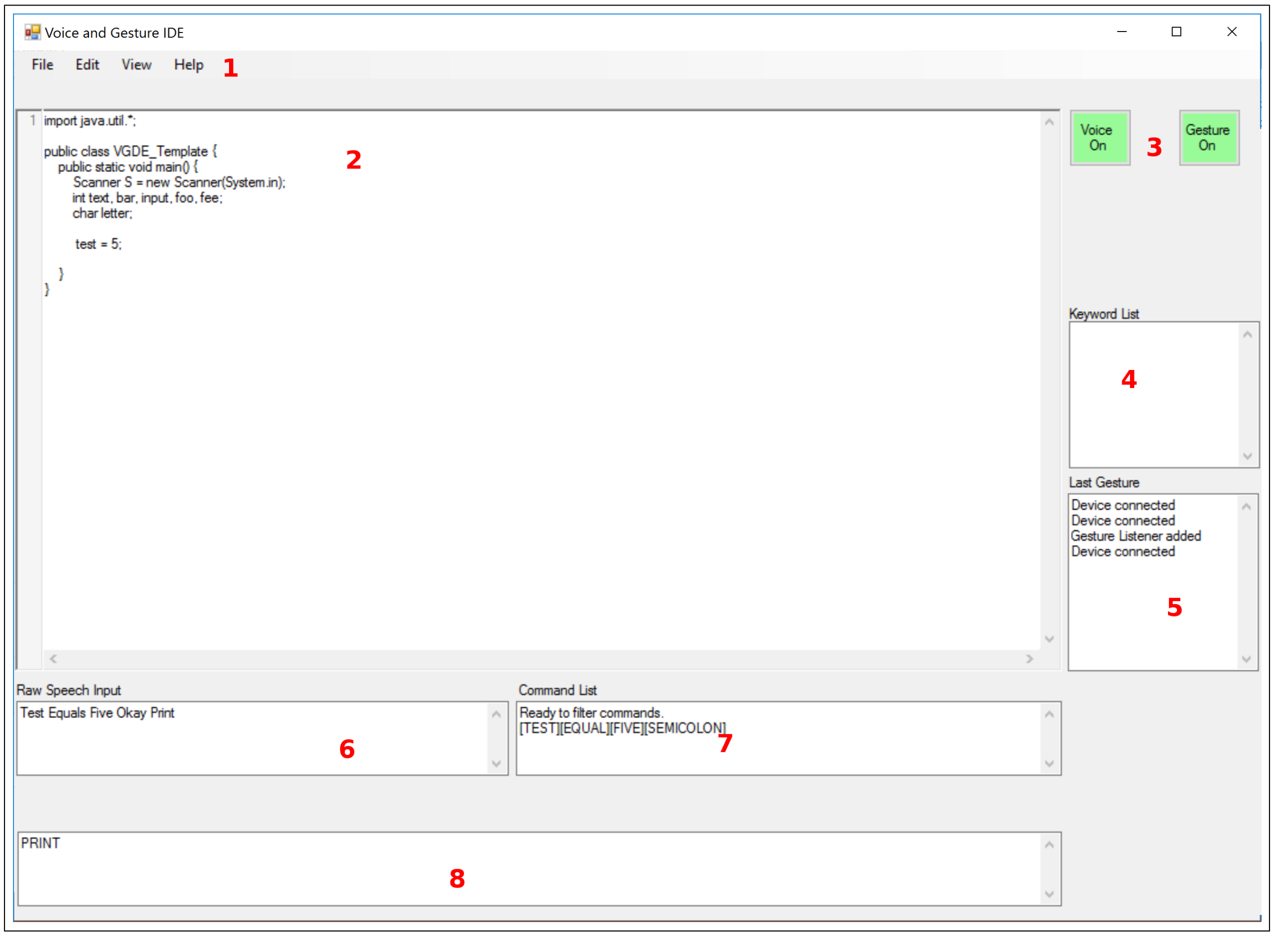
The *Command Parser Thread* is, like all other threads in this system, spawned by the *Main Component*. This thread waits until the *Command List* is not empty. All commands, regardless of whether they are voice or gesture based, are added to the *Command List*. The thread then removes the first item in the *Command List* to process it. This command is passed to the appropriate function for processing, dependent on what category and type of command it is. These functions determine how to execute the commands or whether code should be outputted to the code area in the *User*

*Interface*. The *Keyword* in the *Command list* keeps track of what category each command falls under, so processing the command can be accomplished faster. For example, the parser first determines if the command is code output. If it is, then the text can be immediately processed and printed to the code area textbox.

## Development Environment

The *Development Environment* of the *VGDE* encompasses all other features of the software. Figure 2.10 shows an image of the user interface for the program, with several components labeled. The current layout of the program is designed to provide as much information as possible for developers. Item 1 is the menu bar for the program, where users may manually execute environment commands, such as file functions and simple editting features. Item 2 shows the code area, which is a Rich Text Area where the user is able to create, edit, and navigate their code. The user can save this code to a file or load a text file into this area through voice commands or the menu bar, found at Label 1. Item 3 contain buttons to mute or pause the microphone, if the user would like to pause voice command recognition, . This can also be accomplished through voice command. The only voice commands recognized while on mute is the command to resume voice commands. Item 4 contains a list of available identifiers for that program - however this feature is not currently implemented. Item 5 contains status information on the Leap Motion device and gesture listeners. Item 6 displays the Raw Speech Input, which represents all recognized utterances from the user. Item 7 contains the Parsed Commands List, a list of all commands recognized from the user's Raw Speech Input. Lastly, item 8 is the miscellaneous information output. This textbox displays other information relevant to to VGDE software, such as the current contents of the *prevWords* list.

1. Menu Bar
2. Code Area
3. Voice and Gesture Controls
4. Available Identifiers (not implemented)
5. Last Gesture Device / Listener Status
6. Raw Speech Input: All recognized utterances
7. Parsed Commands List
8. Miscellaneous Information Output: Current *prevWords* list



```
import java.util.*;

public class VGDE_Template {
    public static void main() {
        Scanner S = new Scanner(System.in);
        int text, bar, input, foo, fee;
        char letter;

         test = 5;

    }
}
```

```
Device connected
Device connected
Gesture Listener added
Device connected
```

```
Test Equals Five Okay Print
```

```
Ready to filter commands.
[TEST][EQUAL][FIVE][SEMICOLON]
```

```
PRINT
```

Figure 2.10: Image of VGDE user interface, with components labeled 1-8.

# CHAPTER III: EVALUATION AND RESULTS

The *Voice and Gesture Development Environment* has been evaluated in two main categories: accuracy in recognizing voice commands and accuracy in recognizing gesture commands. Each of these evaluation categories has several sets of commands that we tested, which were executed three times for each variation of the test.

## 3.1 Evaluation of the Accuracy of Voice Commands

The *Voice Component* of this system relies on accurately recognizing what the user says. There are a few factors that contribute to correctly recognizing words. We hypothesize microphone quality is one of the factors. We believe lower quality microphones, such as built-in laptop microphones, will produce more errors than higher quality microphones. We tested the *VGDE* on two different microphones: the *RealTek High Definition Audio* microphone integrated into the *ASUS Zenbook Pro UX501VW*[3] laptop, and the *Blue Microphone Snowball*[4]. The second factor considered in evaluating the accuracy of voice commands is how long of a pause the user allows between each command.

The test criteria consisted of dictating a simple java program, which was adapted from a textbook on Java [11]. Below is listed the commands spoken to create the test code. Figure 3.11 displays the output from these spoken commands in the *VGDE* software, while figure 3.12 displays the full program, including the headers and miscellaneous code loaded by the *VGDE* software. For the first test, the code was dictated with a one to two second pause between each command. A second test was attempted with a pause of less than one second; however, the test was unable to be completed due to an excessive amount of errors. Table 3.15 displays the list of commands spoken to generate the test program. The order of the commands read from left to right, top to bottom.

Table 3.15: A list of voice commands spoken for evaluation test.

| | | | |
|---|---|---|---|
| Test | Equals | Get Int | Okay |
| Bar | Equals | One | Okay |
| Return | While | Bar | Less than or equal to |
| Test | Divided by | Two | Do |
| Bar | Equals | Bar | Times |
| Two | Okay | End while | Return |
| While | Bar | Greater than | Zero |
| Do | If | Test | Less than |
| Bar | Then | Print | Zero |
| End print | End if | Else | Print |
| One | End print | Test | Equals |
| Test | Minus | Bar | Okay |
| End else | Return | Bar | Equals |
| Bar | Divided by | Two | Okay |
| End while | | | |

```
test = S.nextInt();
bar = 1;

while( bar <= test / 2){
    bar = bar * 2;
}

while( bar > 0){
    if( test < bar){
        System.out.print( 0);
    }
    else {
        System.out.print( 1);
        test = test - bar;
    }

    bar = bar / 2;
}
```

Figure 3.11: Voice Command Test 1: Convert Integer to Binary

```
import java.util.*;

public class VGDE_Template {
    public static void main() {
        Scanner S = new Scanner(System.in);
        int text, bar, input, foo, fee;
        char letter;

        test = S.nextInt();
        bar = 1;

        while( bar <= test / 2){
            bar = bar * 2;
        }

        while( bar > 0){
            if( test < bar){
                System.out.print( 0);
            }
            else {
                System.out.print( 1);
                test = test - bar;
            }

            bar = bar / 2;
        }
    }
}
```

Figure 3.12: Full Test Code - Including headers and template

## 3.2 Evaluation of the Accuracy of Gesture Commands

There are a few factors which contribute or negatively impact the accuracy of the Leap Motion device. One of these factors is whether the surface of the device is clean and free of marks or obstructions. These can include smudges, fingerprints, or liquid. An unclean surface can impact the infrared and other light signals that the devices uses to detect objects and movement. Another factor affecting the accuracy of gesture recognition is the distance between the Leap Motion device and the users hands. The device has a range of two feet in all directions.

Another issue with correctly recognizing gesture commands is the actual hand tracking model of the Leap Motion. Occasionally, the model will twist and become stuck in an unrecognizable pose, which interferes with gesture recognition. A solution to this to reset the hand model. The user does this by closing their hand into a fist while above the Leap Motion, then opening it again with their fingers spread. After this, the hand model returns to normal and the user is able to use it again without error for some time. Table 3.16 displays the sequence of gesture commands performed for this test. The order of gestures read from left to right, top to bottom.

Table 3.16: A list of gesture commands executed for evaluation test.

| | |
|---|---|
| 1-Finger Clockwise Circle | 1-Finger Clockwise Circle |
| 1-Finger Clockwise Circle | 1-Finger Counter-Clockwise Circle |
| 1-Finger Counter-Clockwise Circle | 2-Fingers Right to Left Swipe |
| 1-Finger Clockwise Circle | 1-Finger Right to Left Swipe |
| 1-Finger Counter-Clockwise Circle | 1-Finger Left to Right Swipe |
| 1-Finger Clockwise Circle | 2-Finger Left to Right Swipe |
| 1-Finger Right to Left Swipe | 1-Finger Downward Swipe |
| 1-Finger Counter-Clockwise Circle | 1-Finger Counter-Clockwise Circle |
| 3-Finger Downward Swipe | |

## 3.3 Results of the Evaluations

Our evaluation of the Voice and Gesture Development Environment shows that allowing for a one to two second pause between voice commands allows users to create programs by voice with very few errors, compared to allowing for less than one second pause between commands. The test consisted of dictating a simple Java program on two different microphone, at two different speeds. The first speed consisted of a less than one second pause between each voice command, while the second allowed for one to two pauses between each voice command. The results for these tests can be seen in figure 3.17. Although we predicted that the *Blue Snowball* microphone would produce better results than an integrated laptop microphone, both microphones have proven to perform well, with low number of errors.

Table 3.17: Results of the Voice Command evaluation on microphone and timing accuracy.

| **Microphone** | **<1 sec. pause errors** | **1-2 sec. pause errors** |
|---|---|---|
| **Laptop** | | |
| *Test 1* | >20 | 2 |
| *Test 2* | >20 | 1 |
| *Test 3* | >20 | 0 |
| *Average* | >20 | 1 |
| **Snowball** | | |
| *Test 1* | >20 | 2 |
| *Test 2* | >20 | 0 |
| *Test 3* | >20 | 1 |
| *Average* | >20 | 1 |

To demonstrate the accuracy of gestures, we evaluated the accuracy of gesture recognition using a series of seventeen commands. The test was conducted three times, with a one to two second pause between each gesture. Table 3.18 displays the results of these tests. There are three cells indicating the different results evaluated. The first, labeled *Hand Resets* indicates the number of times that the hand position had to be reset. This is done by closing the hand into a fist, then extending the fingers, then continuing on with the next gesture in the test. The next, labeled *False Positives* indicates when the device incorrectly recognizes a different gesture than the user intended. Lastly, the *False Negatives* indicates the device did not recognize the user's gesture as any valid gesture. Both *False Positives* and *False Negatives* require the user re-execute their last command until it is correctly recognized.

Table 3.18: The results of the Gesture Command evaluation on accuracy.

| | **Hand Resets** | **False Positives** | **False Negatives** |
|---|---|---|---|
| **Test 1** | 1 | 0 | 3 |
| **Test 2** | 5 | 7 | 6 |
| **Test 3** | 2 | 1 | 8 |
| **Average** | 2.67 | 2.67 | 5.67 |

For the best experience using the *VGDE* software, users should allow for a one to two second pause between voice commands. The Leap Motion device should be clean and free of marks.

# CHAPTER IV: FUTURE WORK

One area in which the *VGDE* could be further developed is the optimization of how voice commands for the output of code are parsed. Currently, when the software parses voice commands there is often a delay in fully recognizing code commands that also occur within longer code commands. This could be remedied by outputting the partially recognized code, then updating or replacing the most recently printed code if the full command is different. For example, in the current version of *VGDE*, if the user uses voice commands to say “three greater than four”, the parser will not print the “greater than” symbol until the user says “four”. This is because the parser is waiting to determine whether the user wants to output “greater than or equal to”. This process could be optimized by simply outputting the greater than symbol, and if the user continues with “or equal to”, it would update the last outputted code. This optimization would improve the users experience with the software by allowing them to see code immediately after using a voice command.

In addition to creating better parsing capabilities, improving the semantics of the *Voice Component* would be beneficial. One way in which the semantics could be improved is to have a wider variety of means to execute commands. Providing more means for the user to speak naturally and comfortably could increase the usability of the software. Although the software as it is aims to have a relatively natural method of speech, there is room for improvement. Creating better semantics could also involve the software learning the users speech patterns and adapting.

Another area that could be further developed is the addition of more features supported by the software. Creating support for more programming languages can be accomplished without re-writing the entire software. This could be easily done through the *Grammar Class* and *Keyword Class*. A means to change the current language could be implemented as a settings window or menu option in the user interface. By providing support for more languages, this increases the usability of this program. Currently, the only language supported is Java. If more programming languages were available, users would be able to program for a wider range of programming needs.

In addition to providing a wider range of programming languages, increasing the amount of syntax recognized and processed would allow for more complex programs able to be created using this software.

Improvements are also possible through the user interface. Currently only one document at a time can be opened in the program. By allowing users to open more documents at the same time in different tabs, a better work experience can be provided. Source code for software is often separated into multiple files for easier portability. The user interface is a major contributing factor to a user's overall satisfactory experience with software. The user's satisfaction is as important as the usability of the software.

Combining the *VGDE* with an open source integrated development environment (IDE) is another aspect of improving this work. An IDE allows a user to edit source code and compile it into an executable program. This software also contains error checking capabilities that would be very beneficial to the user. More support for incorporating voice or gesture commands to interact with the IDE software would also be a beneficial addition.

To further contribute to the accessibility for manually impaired programmers, the *Gesture Component* could be either replaced or supplemented with an *Eye Gaze Component*. This new feature could allow users to navigate the program by directing the location of the mouse cursor with their eyes. This could reduce the need for manual operation of keyboard and mouse further.

Lastly, future work based on the *VGDE* system could be the development of a new programming language based on voice and gestures. This could be developed based on this thesis to create a compiler which uses voice and gesture commands as a language in and of itself. This would replace the support for programming in Java, C#, or any other programming language. Text-based languages were developed for traditional input devices, such as keyboards and computer mice. While providing a means to program these languages by voice is beneficial for environments where these languages are necessary, creating a language tailored to voice and gestures would be more efficient.

# CHAPTER V: CONCLUSION

A keyboard-free means of programming increases accessibility to programming for manually impaired programmers, as well as provides a new way for all programmers to create software. There have been several approaches from various authors to provide a means of reducing keyboard and mouse use were explored.

Our software improves on these past works in several different ways. Keyboard and mouse usage is reduced through the *Voice and Gesture Development Environment*. By allowing users to create Java code through voice commands, keyboard usage is reduced. Voice commands also allow the user to edit their code, navigate through their code, and interact with the program. Users are able to use hand gestures to execute a limited selection of commands, which reduces mouse usage while programming. These commands include code navigation and editing. The system is situated in a development environment, where the user can save and open their programming files to work on. Users can open code files even if the document was not created and previously edited in the *VGDE* software. The user may also manually edit their code at any time in this software, which prevents the user from being limited by the voice and gesture commands available.

Although there are many contributions made by this thesis, there are still several issues. The software currently has difficulties with recognizing voice and gesture commands if certain conditions are not met. Best results for recognizing voice commands require the user to allow one to two seconds between speaking voice commands. Gesture commands are typically not recognized accurately when the surface of the Leap Motion device is not well cleaned and free of any marks. In addition to voice and gesture recognition issues, there are limitations in regards to the complexity of programs that can be created through the *VGDE*. The limited semantics that are recognized by the *Voice Component* can cause difficulties when creating more complex programs.

These limitations can be overcome with further expansion of the coding voice commands available to the user and the optimization of accuracy for recognizing both voice and gesture commands.

This research has the potential to change how people work within the industry by providing accessibility, extending the length of programming careers by providing a support for the manually impaired, and providing a fresh approach for programming for general users.

## REFERENCES

[1] K. Mohamed Ali and B.W.C. Sathiyasekaran. Computer professionals and carpal tunnel syndrome (cts). *International Journal of Occupational Safety and Ergonomics*, 12(3):319–325, 2006. PMID: 16984790.

[2] Stephen C. Arnold, Leo Mark, and John Goldthwaite. Programming by voice, vocalprogramming. In *Proceedings of the Fourth International ACM Conference on Assistive Technologies*, Assets '00, pages 149–155, New York, NY, USA, 2000. ACM.

[3] ASUS. Asus zenbook pro. Website.

[4] Blue. Blue microphone - products - snowball. Website.

[5] Denis Delimarschi, George Swartzendruber, and Huzefa Kagdi. Enabling integrated development environments with natural user interface interactions. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 126–129, New York, NY, USA, 2014. ACM.

[6] Thomas J. Hubbell, David D. Langan, and Thomas F. Hain. A voice-activated syntax-directed editor for manually disabled programmers. In *Proceedings of the 8th International ACM SIGACCESS Conference on Computers and Accessibility*, Assets '06, pages 205–212, New York, NY, USA, 2006. ACM.

[7] Arun Kumar, Sheetal K. Agarwal, and Priyanka Manwani. The spoken web: Software development and programming through voice. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 1371–1374, New York, NY, USA, 2010. ACM.

[8] Inc. Leap Motion. Leap motion v2 tracking. Website.

[9] Microsoft. Microsoft speech platform. Website.

[10] Jean K. Rodriguez-Cartagena, Andrea C. Claudio-Palacios, Natalia Pacheco-Tallaj, Valerie Santiago González, and Patricia Ordonez-Franco. The implementation of a vocabulary and grammar for an open-source speech-recognition programming platform. In *Proceedings of the 17th International ACM SIGACCESS Conference on Computers &#38; Accessibility*, ASSETS '15, pages 447–448, New York, NY, USA, 2015. ACM.

[11] Robert Sedgewick and Kevin Wayne. Introduction to programming in java. Website, January 2017.

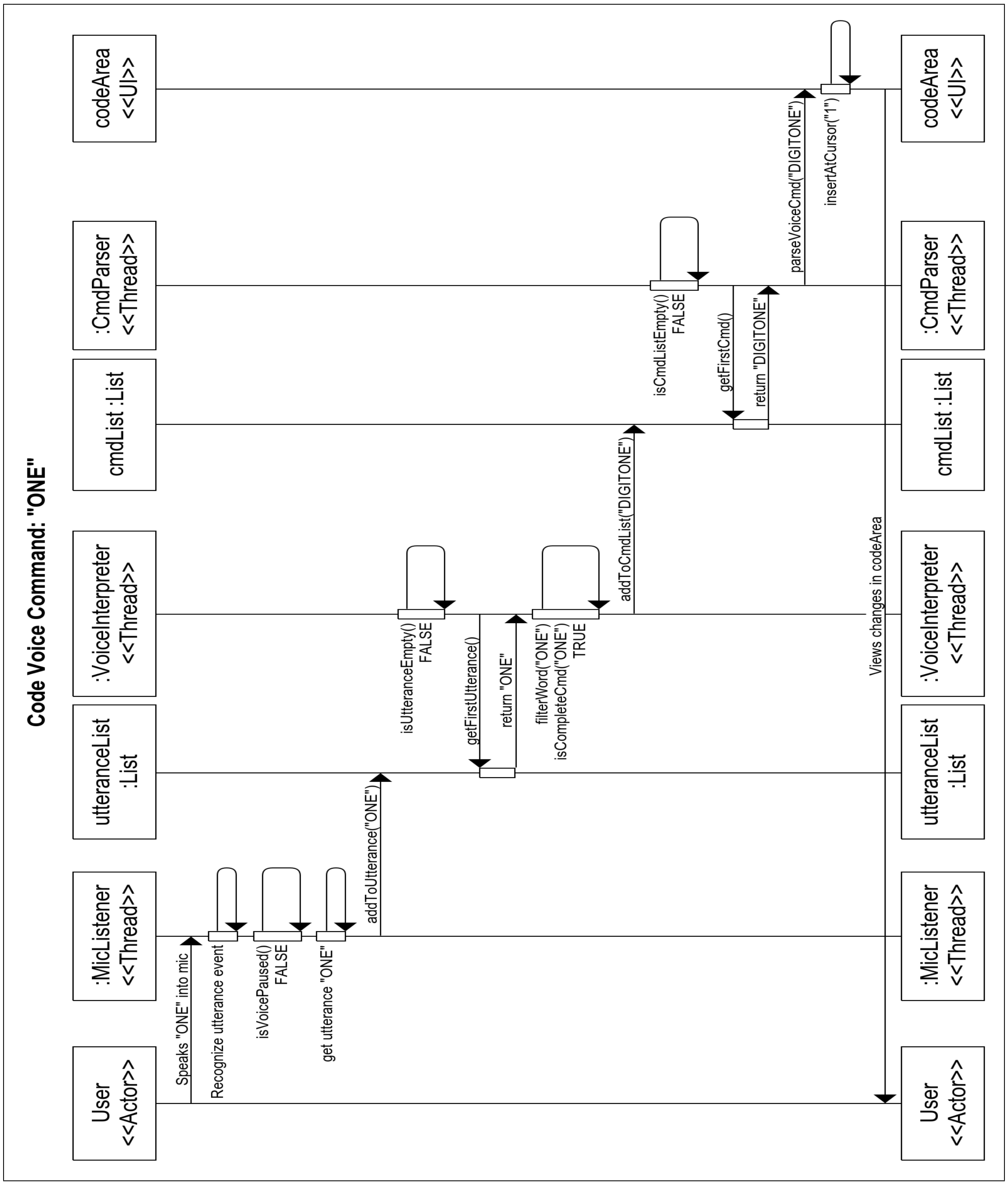


Figure 6.13: Sequence Diagram of Voice Programming Command “ONE”

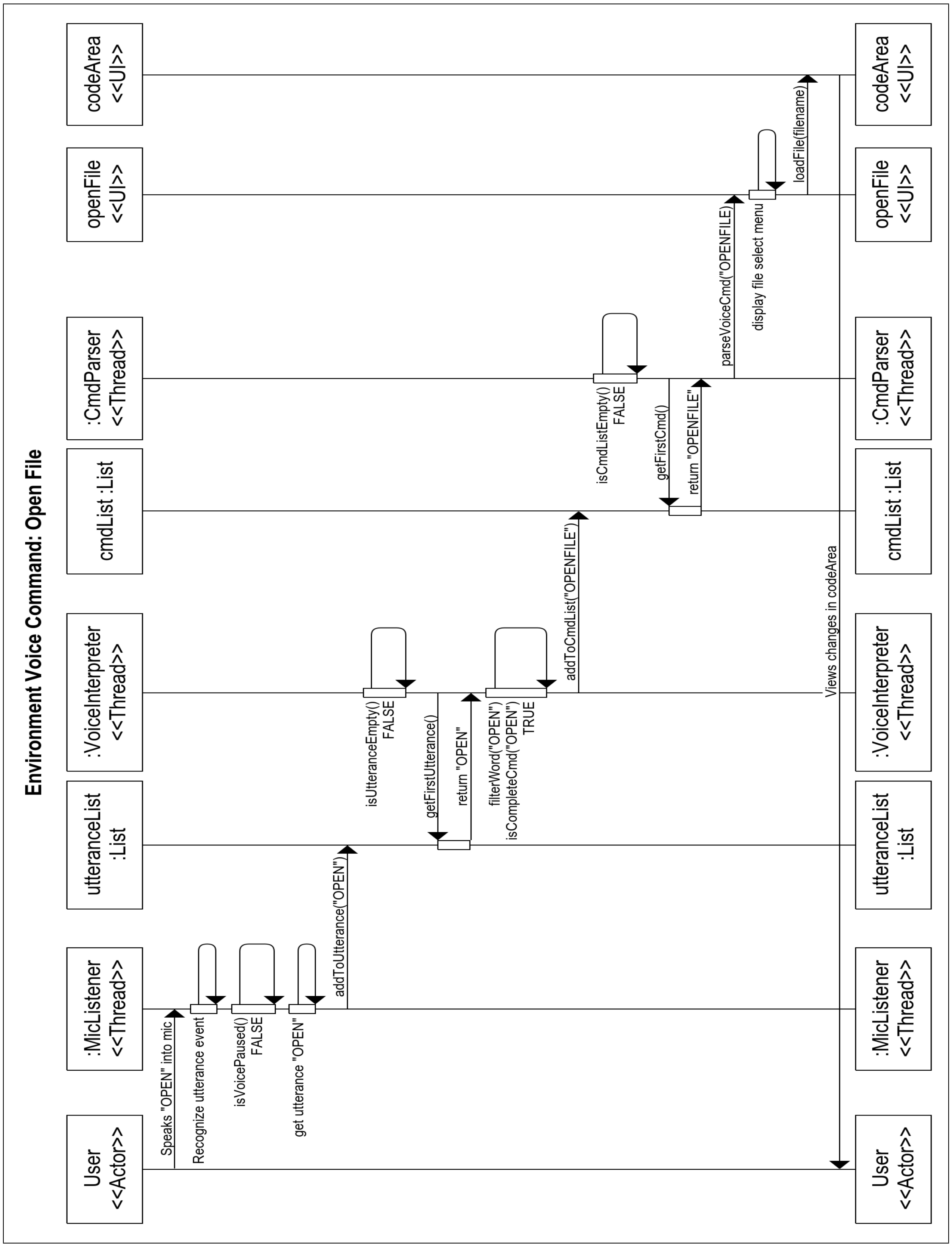


Figure 6.14: Sequence Diagram of Voice Environment Command “Open File”

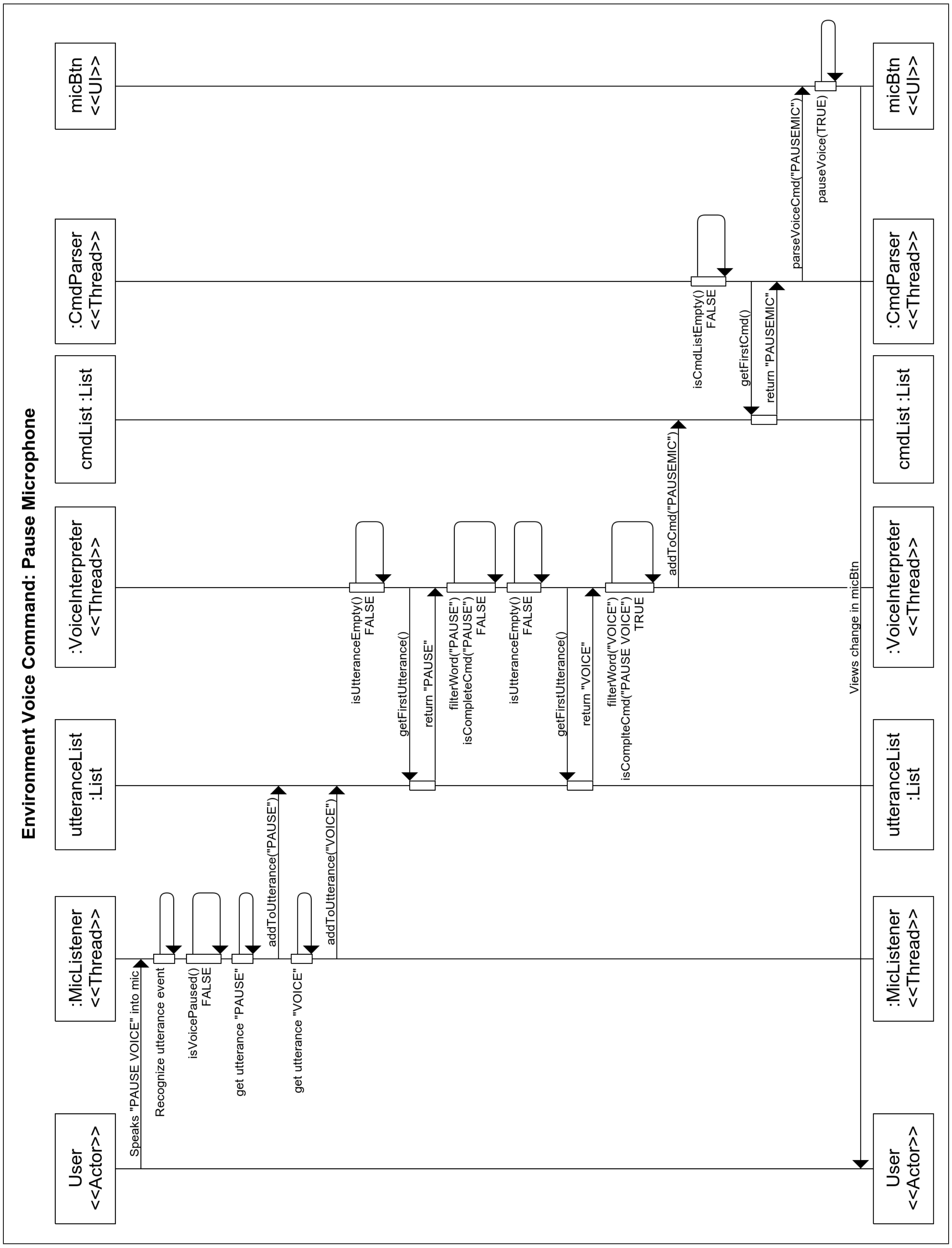


Figure 6.15: Sequence Diagram of Voice Environment Command “Pause Microphone”

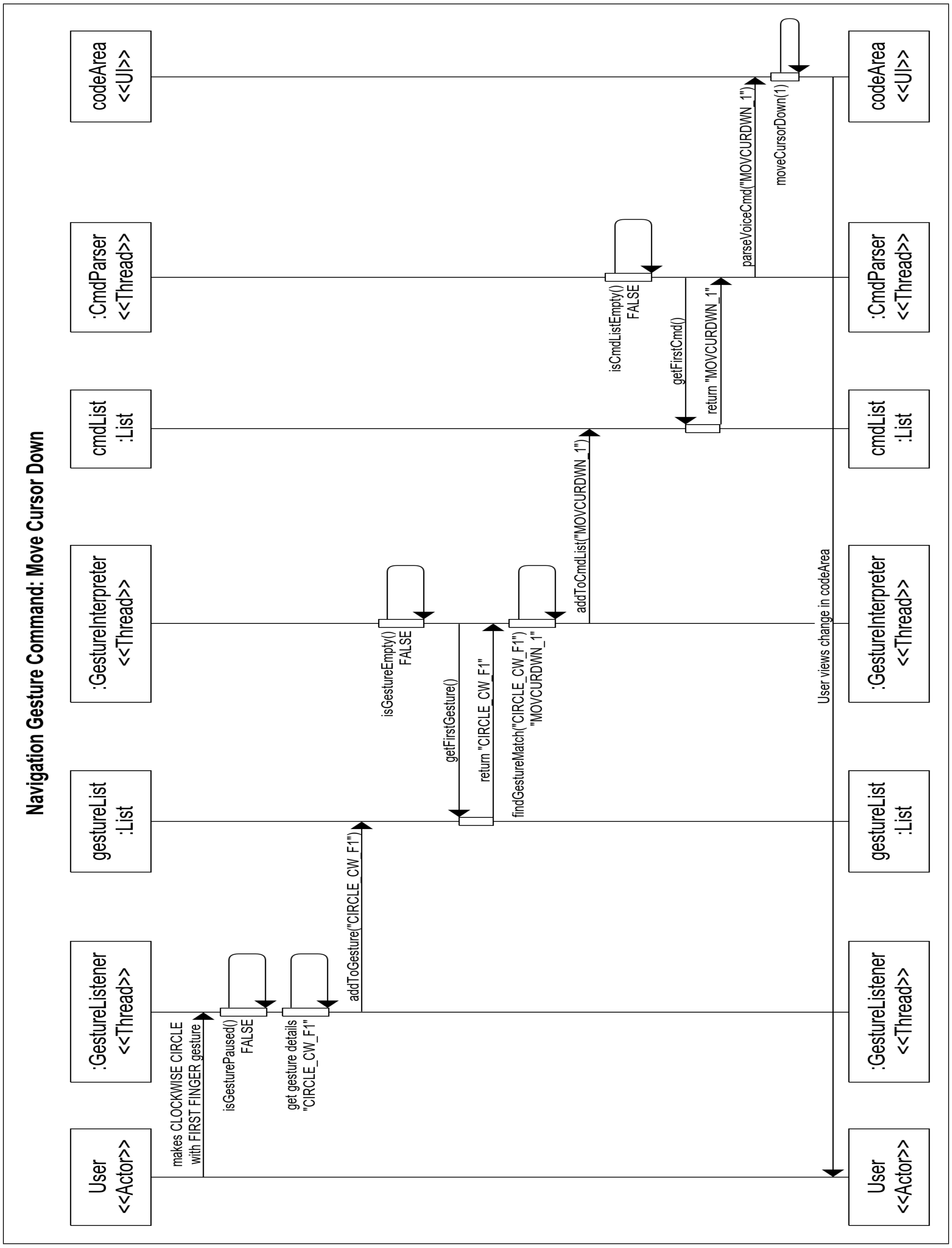

Navigation Gesture Command: Move Cursor Down
User <<Actor>>
:GestureListener <<Thread>>
gestureList :List
:GestureInterpreter <<Thread>>
cmdList :List
:CmdParser <<Thread>>
codeArea <<UI>>
makes CLOCKWISE CIRCLE with FIRST FINGER gesture
isGesturePaused() FALSE
get gesture details "CIRCLE_CW_F1"
addToGesture("CIRCLE_CW_F1")
isGestureEmpty() FALSE
getFirstGesture()
return "CIRCLE_CW_F1"
findGestureMatch("CIRCLE_CW_F1") "MOVCURDWN_1"
addToCmdList("MOVCURDWN_1")
isCmdListEmpty() FALSE
getFirstCmd()
return "MOVCURDWN_1"
parseVoiceCmd("MOVCURDWN_1")
moveCursorDown(1)
User views change in codeArea

Figure 6.16: Sequence Diagram of Gesture Navigation Command “Move Cursor Down”